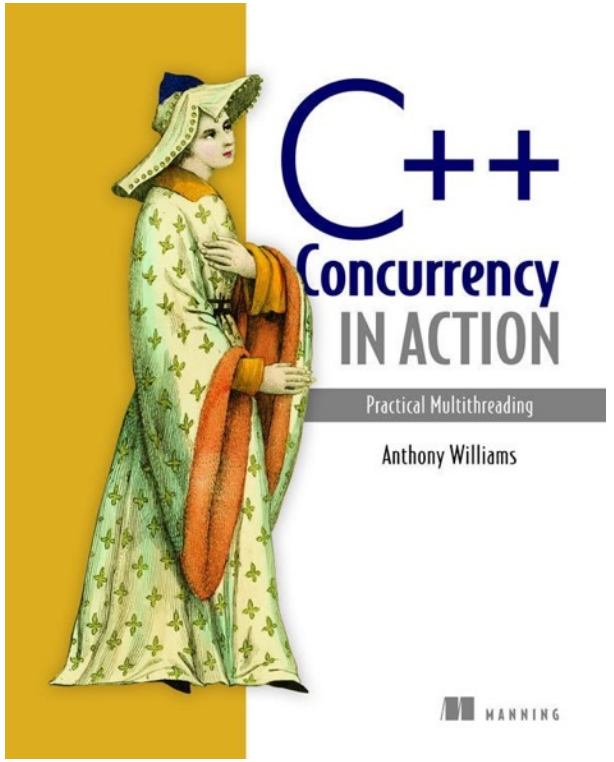
C++ Korea
C++ Concurrency in Action Study

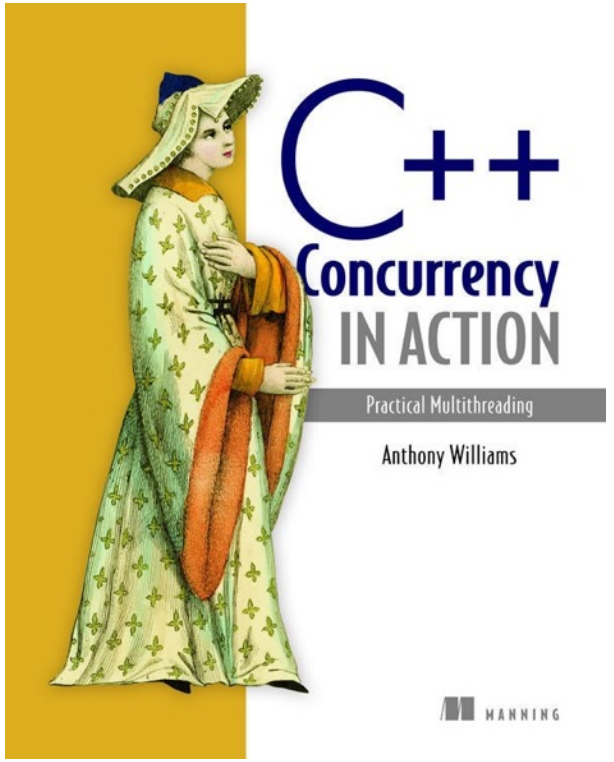


Chapter 07 Designing lock free concurrent data structure

- Definitions and consequences
- Implementation for lock-free stack, lock-free queue
- Guidelines for lock-free

C++ Korea 최두선 (returns@lycos.co.kr)

Definition and consequences



Blocking

- mutex 나 condition_value 를 사용하는 알고리즘과 자료구조
- 어플리케이션이 다른 쓰레드가 수행을 완료하기 전까지 실햄을 멈춤

- 실행을 중지하고 운영체제가 다른 프로세스를 실행할수 있다

- A process that is blocked is one that is waiting for some event

non-Blocking

- block 라이브러리 함수를 사용하지 않은 자료구조와 알고리즘

blocking, non-blocking ?? sync, async

- Blocking and synchronous mean the same thing: you call the API, it hangs up the thread until it has some kind of answer and returns it to you.
- Non-blocking means that if an answer can't be returned rapidly, the API returns immediately with an error and does nothing else. So there must be some related way to query whether the API is ready to be called (that is, to simulate a wait in an efficient way, to avoid manual polling in a tight loop).
- Asynchronous means that the API always returns immediately, having started a "background" effort to fulfil your request, so there must be some related way to obtain the result.

Implementation of a spin-lock mutex using std::atomic_flag

```
class spinlock_mutex
{
    std::atomic_flag flag;
public:
    spinlock_mutex() : flag( ATOMIC_FLAG_INIT )
    {
    }

    void lock()
    {
        while( flag.test_and_set( std::memory_order_acquire ) );
    }

    void unlock()
    {
        flag.clear( std::memory_order_release );
    }
};
```

- non-blocking
- lock-free 는 아니다.!!
  - mutex 가 존재
  - 한번에 하나의 쓰레드만 lock 을 획득 가능

Lock-free Data Structures

- 하나 이상의 쓰레드가 동시에 자료구조에 접근 가능
- 그러나 같은 연산을 수행하는것은 가능하지 않음
  - 하나의 쓰레드가 push() 를 수행하고 다른 하나는 pop() 을 할수 있음
  - 두 쓰레드가 동시에 새로운 아이템을 동시에 push()는 불가능
- 자료구조에 접근하는 쓰레드중 하나가 스케쥴러에 의하여 연산 수행 도중 정리 되어도 다른 쓰레드는 정지된 쓰레드를 대기 하지 않고 그 자신의 연산을 완료 가능
- 자료구조에 compare/exchange 연산을 사용하여 알고리즘을 구현
  - compare/exchange 가 성공하였으면 데이터 변경됨
  - compare/exchange 가 실패하였으면 예전으로 돌아가서 다시 연산을 수행

Lock-free Data Structures

- 다른 쓰레드가 정지하였을때
  - compare/exchange 가 성공하는 코드는 lock-free 라고 할수 있다.
  - compare/exchange 가 실패되는 코드는 non-blocking 이지만 lock-free 는 아니다. -> spin lock

- 다른 쓰레드가 '잘못된' 타이밍으로 연산을 수행하려고 한다면 그 쓰레드는 계속 대기 할수 있다. 이런 문제를 starvation 이라고 한다.

- 이런 문제를 피할수 있는 자료구조는 wait-free 자료구조이다.

wait-free data structures

- lock-free 자료구조에 추가적으로 다른 쓰레드의 행동과 무관하게 그 자신의 연산을 정해진 스텝안에 완료

- 제한되지 않은 수의 재 시도를 하는 알고리즘은 wait-free 가 아니다.

- wait-free 자료구조를 작성하기란 매우 어렵다.

The pros and cons of lock-free data stucture

- 장점
  - 최대 concurrency 가 가능
  - 견고함(robustness)
- 단점
  - 알고리즘이 많이 복잡 (하나의 쓰레드만 접근 하더라도 많은 연산을 수행)
  - 전체적인 성능 하락
    - 더 많은 연산을 수행
    - atomic 연산은 non-atomic 연산보다 느림
    - atomic 변수를 접근하는 쓰레드 사이의 데이터를 동기화 시킴
    - chapter08 에서 이야기할 cache ping pong 문제가 발생

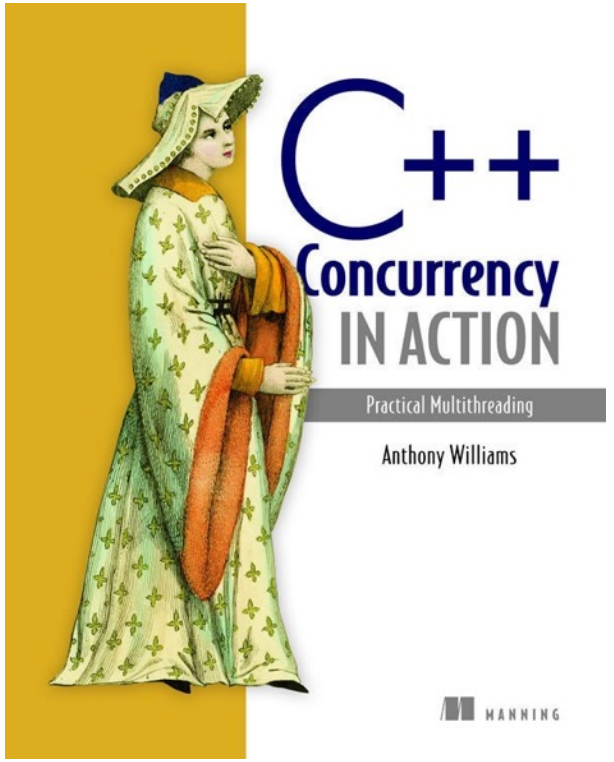
live lock

- 두 쓰레드가 락의 해제와 획득을 반복하는 상태
- lock-free 에서는 락이 없다…?
- 그러나..
  - 두 쓰레드가 각자 자료구조를 변경하려고 시도할때 발생..
  - 각 쓰레드에서 다른 쓰레드에서 수행한 변경 때문에 연산을 다시 수행한다.
  - 그러면 다시 연산을 수행하고 다시 다른 쓰레드에서 수행한 변경 때문에 연산을 다시 수행한다.
  - 이때문에 라이브락이 발생
- wait-free 코드는 라이브락이 발생하지 않는다.
  - 연산을 수행할때 정해진 스텝 안에 완료되기 때문에

Definitions and Consequences

- lock-based 자료구조와 lock free 자료구조 둘다 반영전후 성능을 확인해 보아야 한다.

Writing a thread-safe stack without locks



Hazard pointer

- Each reader thread owns a single-writer/multi-reader shared pointer called "hazard pointer." When a reader thread assigns the address of a map to its hazard pointer, it is basically announcing to other threads (writers), "I am reading this map. You can replace it if you want, but don't change its contents and certainly keep your deleteing hands off it."

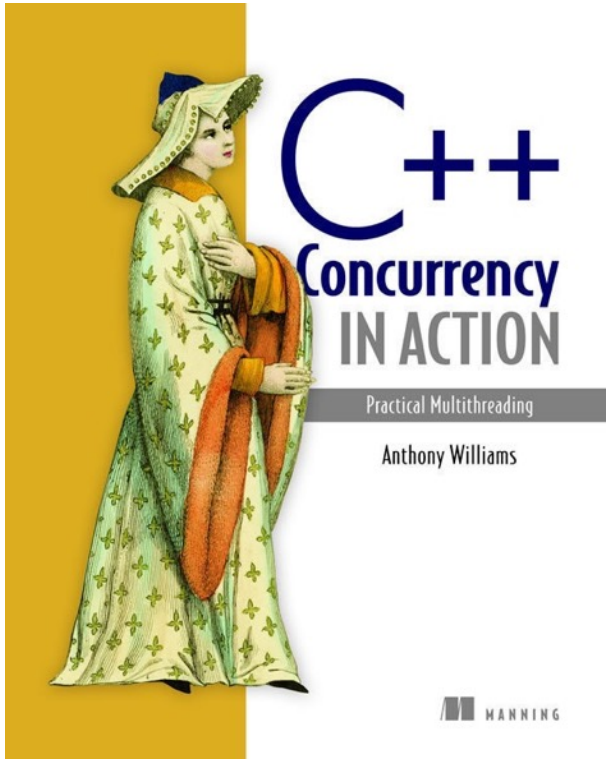
- — Andrei Alexandrescu and Maged Michael, Lock-Free Data Structures with Hazard Pointers[2]

- In a hazard-pointer system, each thread keeps a list of hazard pointers indicating which nodes the thread is currently accessing. (In many systems this "list" may be provably limited to only one[1][2] or two elements.) Nodes on the hazard pointer list must not be modified or deallocated by any other thread.

- Each reader thread owns a single-writer/multi-reader shared pointer called "hazard pointer." When a reader thread assigns the address of a map to its hazard pointer, it is basically announcing to other threads (writers), "I am reading this map. You can replace it if you want, but don't change its contents and certainly keep your deleteing hands off it."

- — Andrei Alexandrescu and Maged Michael, Lock-Free Data Structures with Hazard Pointers[2]
- When a thread wishes to remove a node, it places it on a list of nodes "to be freed later", but does not actually deallocate the node's memory until no other thread's hazard list contains the pointer. This manual garbage collection can be done by a dedicated garbage-collection thread (if the list "to be freed later" is shared by all the threads); alternatively, cleaning up the "to be freed" list can be done by each worker thread as part of an operation such as "pop" (in which case each worker thread can be responsible for its own "to be freed" list).

Writing a thread-safe queue without locks



A single-producer, single-consumer lock free queue

- 한번에 하나의 쓰레드만 push() 나 pop() 을 호출하면 문제없도록 구현
  - push() 가 여러 쓰레드에서 호출되지 않음
  - pop() 역시 여러 쓰레드에서 호출되지 않음
  - push() 와 pop() 은 같은 호출 가능

- queue 에서의 공유되는 데이터
  - head 와 tail 포인터

- 공유되는 데이터를 변경시 mutex 를 사용하기 보다는 atomic 연산 사용

A single-producer, single-consumer lock free queue

```
template<typename T>class lock_free_queue
{
private:
    struct node
    {
        std::shared_ptr<T> data;
        node*              next;

        node() : next(nullptr)
        {

        }
    };

    std::atomic<node*> head;
    std::atomic<node*> tail;

    node* pop_head()
    {
        node* const old_head=head.load();

        if( old_head == tail.load() )
        {
            return nullptr;
        }

        head.store( old_head->next );

        return old_head;
    }

public:
+-- 42 lines: lock_free_queue(): head(new node),--------------
};
```

A single-producer, single-consumer lock free queue

```
template<typename T>class lock_free_queue
{
private:
+-- 27 lines: struct node-----------------------------------------------------

public:
    lock_free_queue(): head(new node), tail(head.load())
    {}

    lock_free_queue( const lock_free_queue& other ) = delete;
    lock_free_queue& operator=( const lock_free_queue& other ) = delete;

    ~lock_free_queue()
    {
        while( node* const old_head = head.load() )
        {
            head.store( old_head->next );
            delete old_head;
        }
    }

    std::shared_ptr<T> pop()
    {
+---  12 lines: node* old_head = pop_head();---------------------------------
    }

    void push( T new_value )
    {
+---   7 lines: std::shared_ptr<T> new_data( std::make_shared<T>( new_value ) );--
    }
};
```

A single-producer, single-consumer lock free queue

```
template<typename T>class lock_free_queue
{
private:
+-- 27 lines: struct node-----------------------------------------------------

public:
+--- 15 lines: lock_free_queue(): head(new node), tail(head.load())------------
    std::shared_ptr<T> pop()
    {
        node* old_head = pop_head();

        if( !old_head )
        {
            return std::shared_ptr<T>();
        }

        std::shared_ptr<T> const res( old_head->data );

        delete old_head;

        return res;
    }

    void push( T new_value )
    {
+---  7 lines: std::shared_ptr<T> new_data( std::make_shared<T>( new_value ) );-
    }
};
```

A single-producer, single-consumer lock free queue

```
template<typename T>class lock_free_queue
{
private:
+-- 27 lines: struct node-----------------------------------------------------

public:
+--- 15 lines: lock_free_queue(): head(new node), tail(head.load())----------
    std::shared_ptr<T> pop()
    {
+---- 12 lines: node* old_head = pop_head();---------------------------------
    }

    void push( T new_value )
    {
        std::shared_ptr<T> new_data( std::make_shared<T>( new_value ) );
        node*              p = new node;
        node* const        old_tail = tail.load()

        old_tail->data.swap( new_data );
        old_tail->next = p;
        tail.store( p );
    }
};
```

A single-producer, single-consumer lock free queue

- push()
  - push()를 동시에 호출하는 2개의 쓰레드가 존재
  - 둘다 새로운 dummy 노드로 새로운 노드를 할당
  - 둘다 tail 로 부터 같은 값을 로드
  - 마지막으로 둘다 데이터와 next 포인터를 갱신할때 같은 노드의 데이터 멤버를 둘다 갱신
  - data race 발생

A single-producer, single-consumer lock free queue

	Thread 1		Thread 2
	`push( 'A' );`		`push( 'B' );`
1	`std::shared_ptr<T> new_data( std::make_shared<T>( new_value ) );`		
2	`node*  p = new node;`		
3	`node* const  old_tail = tail.load();`		
4			`std::shared_ptr<T> new_data( std::make_shared<T>( new_value ) );`
5			`node*  p = new node;`
6			`node* const  old_tail = tail.load();`
7	`old_tail->data.swap( new_data );`		
8	`old_tail->next = p;`		
9	`tail.store( p );`		
10			`old_tail->data.swap( new_data );`
11			`old_tail->next = p;`
12			`tail.store( p );`

queue

Node	<- HEAD <- TAIL

Thread 1	
new_data	`A`
p	new node1
old_tail	Node

Thread 2	
new_data	
p	
old_tail	

A single-producer, single-consumer lock free queue

	Thread 1		Thread 2
	`push( 'A' );`		`push( 'B' );`
1	`std::shared_ptr<T> new_data( std::make_shared<T>( new_value ) );`		
2	`node*  p = new node;`		
3	`node* const  old_tail = tail.load();`		
4			`std::shared_ptr<T> new_data( std::make_shared<T>( new_value ) );`
5			`node*  p = new node;`
6			`node* const  old_tail = tail.load();`
7	`old_tail->data.swap( new_data );`		
8	`old_tail->next = p;`		
9	`tail.store( p );`		
10			`old_tail->data.swap( new_data );`
11			`old_tail->next = p;`
12			`tail.store( p );`

queue

Node	<- HEAD <- TAIL

Thread 1	
new_data	`A`
p	new node1
old_tail	Node

Thread 2	
new_data	`B`
p	new node2
old_tail	Node

A single-producer, single-consumer lock free queue

	Thread 1		Thread 2
	`push( 'A' );`		`push( 'B' );`
1	`std::shared_ptr<T> new_data( std::make_shared<T>( new_value ) );`		
2	`node*  p = new node;`		
3	`node* const  old_tail = tail.load();`		
4			`std::shared_ptr<T> new_data( std::make_shared<T>( new_value ) );`
5			`node*  p = new node;`
6			`node* const  old_tail = tail.load();`
7	`old_tail->data.swap( new_data );`		
8	`old_tail->next = p;`		
9	`tail.store( p );`		
10			`old_tail->data.swap( new_data );`
11			`old_tail->next = p;`
12			`tail.store( p );`

queue

Node 'A'
new node1

<- HEAD

<- TAIL Thread1

Thread 1	
new_data	`A`
p	new node1
old_tail	Node

Thread 2	
new_data	`B`
p	new node2
old_tail	Node

A single-producer, single-consumer lock free queue

	Thread 1		Thread 2
	`push( 'A' );`		`push( 'B' );`
1	`std::shared_ptr<T> new_data( std::make_shared<T>( new_value ) );`		
2	`node*  p = new node;`		
3	`node* const  old_tail = tail.load();`		
4			`std::shared_ptr<T> new_data( std::make_shared<T>( new_value ) );`
5			`node*  p = new node;`
6			`node* const  old_tail = tail.load();`
7	`old_tail->data.swap( new_data );`		
8	`old_tail->next = p;`		
9	`tail.store( p );`		
10			`old_tail->data.swap( new_data );`
11			`old_tail->next = p;`
12			`tail.store( p );`

queue

Node 'B'
New Node2

<- HEAD

<- TAIL Thread2

new node1

Thread1

Thread 1	
new_data	`A`
p	new node1
old_tail	Node

Thread 2	
new_data	`B`
p	new node2
old_tail	Node

A single-producer, single-consumer lock free queue

- pop()
  - 동시에 2개의 쓰레드가 호출하면
  - 둘다 head 의 같은 값을 둘다 읽고
  - 둘다 같은 next 포인터에 옛날 값을 덮어쓴다.
  - 둘다 같은 노드 를 찾았다고 생각

A single-producer, single-consumer lock free queue

	Thread 1		Thread 2
	pop()		pop()
	node* old_head = pop_head();		node* old_head = pop_head();

1	node* pop_head()
	{
	  node* const old_head=head.load();
	
	  if(old_head == tail.load())
	  {
	  return nullptr
	  }		

2			node* pop_head()
			{
			  node* const old_head=head.load();
			
			  if(old_head == tail.load())
			  {
			  return nullptr
			  }

3	head.store(old_head->next);
	
	return old_head;
	}		

4			head.store(old_head->next);
			
			return old_head;
			}

queue

Node 'A' <- HEAD
Node2 <- TAIL

Thread 1	
old_head	Node

Thread 2	
old_head	

A single-producer, single-consumer lock free queue

	Thread 1		Thread 2
	pop()		pop()
	node* old_head = pop_head();		node* old_head = pop_head();

1	node* pop_head()
	{
	  node* const old_head=head.load();
	
	  if(old_head == tail.load())
	  {
	  return nullptr
	  }		

2			node* pop_head()
			{
			  node* const old_head=head.load();
			
			  if(old_head == tail.load())
			  {
			  return nullptr
			  }

3	head.store(old_head->next);
	
	return old_head;
	}		

4			head.store(old_head->next);
			
			return old_head;
			}

queue

Node 'A' <- HEAD
Node2 <- TAIL

Thread 1	
old_head	Node

Thread 2	
old_head	Node

A single-producer, single-consumer lock free queue

	Thread 1		Thread 2
	pop()		pop()
	node* old_head = pop_head();		node* old_head = pop_head();

1	node* pop_head()
	{
	  node* const old_head=head.load();
	
	  if(old_head == tail.load())
	  {
	  return nullptr
	  }		

2			node* pop_head()
			{
			  node* const old_head=head.load();
			
			  if(old_head == tail.load())
			  {
			  return nullptr
			  }

3	head.store(old_head->next);
	
	return old_head;
	}		

4			head.store(old_head->next);
			
			return old_head;
			}

A single-producer, single-consumer lock free queue

	Thread 1		Thread 2
	pop()		pop()
	node* old_head = pop_head();		node* old_head = pop_head();

1	node* pop_head()
	{
	  node* const old_head=head.load();
	
	  if(old_head == tail.load())
	  {
	  return nullptr
	  }		

2			node* pop_head()
			{
			  node* const old_head=head.load();
			
			  if(old_head == tail.load())
			  {
			  return nullptr
			  }

3	head.store(old_head->next);
	
	return old_head;
	}		

4			head.store(old_head->next);
			
			return old_head;
			}

A single-producer, single-consumer lock free queue

	Thread 1		Thread 2
	pop()		pop()
	node* old_head = pop_head();		
1			node* old_head = pop_head();

2	if(!old_head)
	{
	  return std::shared_ptr<T>();
	}
	
	std::shared_ptr<T> const res (old_head->data);
	
	delete old_head;
	
	return res;		

3			if(!old_head)
			{
			  return std::shared_ptr<T>();
			}
			
			std::shared_ptr<T> const res (old_head->data);
			
			delete old_head;
			
			return res;

queue

Node2	<- HEAD <- TAIL

Node 'A'

Thread 1	
old_head	Node 'A'
res	Node 'A'

Thread 2	
old_head	Node 'A'
res	Node 'A'

Handling multiple threads in push()

- 마지막으로 둘다 데이터와 next 포인터를 갱신할때 같은 노드의 데이터 멤버를 둘다 갱신
  - 실제 노드 사이에 dummy node 를 추가 하는 방법
  - 데이터 포인터를 atomic 으로 생성하고 compare/exchange 으로 변경하는 방법

Handling multiple threads in push()

- 실제 노드 사이에 dummy node 를 추가 하는 방법
  - 갱신이 필요한 현 tail 노드의 한 부분은 next 포인터이고 이 작업은 atomic 하여야 함
  - 매번 pop 호출때 마다 2개의 노드를 삭제해야 하고 메모리 할당이 2배로 이루어 진다.

Handling multiple threads in push()

- 데이터 포인터를 atomic 으로 생성하고 compare/exchange 으로 변경하는 방법
  - compare/exchange 수행
    - 성공하면 next 포인터를 안전하게 변경가능
    - 실패하면 다른 쓰레드가 data 를 수정하였기 때문에 반복문을 다시 수행

Handling multiple threads in push()

```
void push( T new_value )
{
    std::unique_ptr<T>   new_data( new T( new_value ) );
    counted_node_ptr     new_next;

    new_next.ptr = new node;
    new_next.external_count = 1;

    for(;;)
    {
        node* const  old_tail = tail.load();
        T*           old_data = nullptr;

        if( old_tail->data.compare_exchange_strong( old_data,
                                                    new_data.get() ) )
        {
            old_tail->next = new_next;
            tail.store( new_next.ptr );
            new_data.release();
            break;
        }
    }
}
```

Handling multiple threads in push()

- push() 를 여러 쓰레드에서 수행하면 문제가 발생하지 않음
- push() 와 pop() 를 여러 쓰레드 에서 발생하면 문제 발생
  - pop() 에서 메모리 해제한 부분을 push() 함수 내에서 접근 할수 있음

Handling multiple threads in push()

	Thread 1		Thread 2
	push()		pop()

	`std::unique_ptr<T> new_data(new T(new_value));`
	`counted_node_ptr new_next;`
	`new_next.ptr = new node;`
	`new_next.external_count = 1;`
	`for(;;)`
	`{`
	`  node* const old_tail = tail.load();`
	`  T* old_data = nullptr;`
	
	`  if( old_tail->data.compare_exchange_strong(`
	`  old_data, new_data.get() ) )`
	`  {`		

1			`node* old_head = pop_head();`
			
			`if( !old_head )`
			`{`
			`  return std::shared_ptr<T>();`
			`}`
			
			`std::shared_ptr<T> const res( old_head->data );`
			
			`delete old_head;`
			
			`return res;`

2	`  old_tail->next = new_next;`
	`  tail.store( new_next.ptr );`
	`  new_data.release();`
	`  break;`
	`  }`
	`}`		

queue

Node	<- HEAD <- TAIL

Thread 1	
new_data	
old_tail	

Thread 2	
old_head	

Handling multiple threads in push()

	Thread 1		Thread 2
	push()		pop()

	`std::unique_ptr<T> new_data(new T(new_value));`
	`counted_node_ptr new_next;`
	`new_next.ptr = new node;`
	`new_next.external_count = 1;`
	`for(;;)`
	`{`
	`  node* const old_tail = tail.load();`
	`  T* old_data = nullptr;`
	
	`  if( old_tail->data.compare_exchange_strong(`
	`  old_data, new_data.get() ) )`
	`  {`		

1			`node* old_head = pop_head();`
			
			`if( !old_head )`
			`{`
			`  return std::shared_ptr<T>();`
			`}`
			
			`std::shared_ptr<T> const res( old_head->data );`
			
			`delete old_head;`
			
			`return res;`

2	`  old_tail->next = new_next;`
	`  tail.store( new_next.ptr );`
	`  new_data.release();`
	`  break;`
	`  }`
	`}`		

queue

Node 'A'	<- HEAD <- TAIL

Thread 1	
new_data	'A'
old_tail	Node

Thread 2	
old_head	

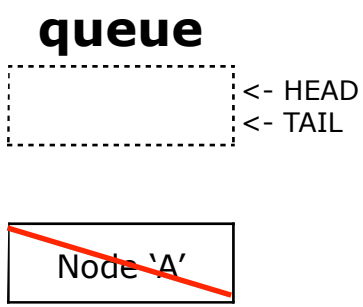
Handling multiple threads in push()

	Thread 1		Thread 2
	push()		pop()

	`std::unique_ptr<T> new_data(new T(new_value));`
	`counted_node_ptr new_next;`
	`new_next.ptr = new node;`
	`new_next.external_count = 1;`
	`for(;;)`
	`{`
	`  node* const old_tail = tail.load();`
	`  T* old_data = nullptr;`
	
	`  if( old_tail->data.compare_exchange_strong(`
	`  old_data, new_data.get() ) )`
	`  {`		

1			`node* old_head = pop_head();`
			
			`if( !old_head )`
			`{`
			`  return std::shared_ptr<T>();`
			`}`
			
			`std::shared_ptr<T> const res( old_head->data );`
			
			`delete old_head;`
			
			`return res;`

2	`  old_tail->next = new_next;`
	`  tail.store( new_next.ptr );`
	`  new_data.release();`
	`  break;`
	`  }`
	`}`		



Thread 1	
new_data	'A'
old_tail	Node

Thread 2	
old_head	Node

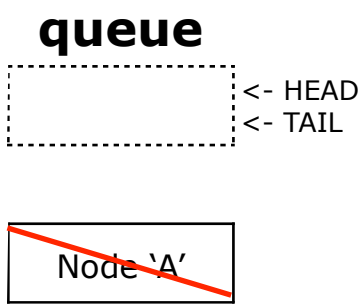
Handling multiple threads in push()

	Thread 1		Thread 2
	push()		pop()

	`std::unique_ptr<T> new_data(new T(new_value));`
	`counted_node_ptr new_next;`
	`new_next.ptr = new node;`
	`new_next.external_count = 1;`
	`for(;;)`
	`{`
	`  node* const old_tail = tail.load();`
	`  T* old_data = nullptr;`
	
	`  if( old_tail->data.compare_exchange_strong(`
	`  old_data, new_data.get() ) )`
	`  {`		

1			`node* old_head = pop_head();`
			
			`if( !old_head )`
			`{`
			`  return std::shared_ptr<T>();`
			`}`
			
			`std::shared_ptr<T> const res( old_head->data );`
			
			`delete old_head;`
			
			`return res;`

2	`  old_tail->next = new_next;`
	`  tail.store( new_next.ptr );`
	`  new_data.release();`
	`  break;`
	`  }`
	`}`		



Thread 1	
new_data	'A'
old_tail	Node

Thread 2	
old_head	Node

Implementing push() for a lock free queue

- 노드를 안전하게 삭제 하려면
  - 각 외부 카운터가 없어질때 노드 구조체 안에 있는 외부 카운터를 감소
  - 외부 카운터가 0 이면 삭제 가능
  - 내부 카운트가 0 이면 외부 카운터는 없음

Implementing push() for a lock free queue

```
template<typename T>
class lock_free_queue
{
private:
    struct node;
    struct counted_node_ptr
    {
        int       external_count;
        node*     ptr;
    };
    std::atomic<counted_node_ptr>     head;
    std::atomic<counted_node_ptr>     tail;
    struct node_counter
    {
        unsigned internal_count:30;
        unsigned external_counters:2;
    };
    struct node
    {
        std::atomic<T*>               data;
        std::atomic<node_counter>     count;
        counted_node_ptr next;
        node()
        {
            node_counter   new_count;

            new_count.internal_count = 0;
            new_count.external_counters = 2;
            count.store( new_count );
            next.ptr = nullptr;
            next.external_count = 0;
        }
    };
```

Implementing push() for a lock free queue

```
struct node_counter
{
    unsigned internal_count:30;
    unsigned external_counters:2;
};
```

- external_counters는 2 bit 만 필요
  - 최대 2개의 카운터만 존재(tail, head)
- internal_count는 30bit
- external_counters + internal_count 의 크기는 머신의 word에 종속
  - 이값들의 갱신은 atomic 하게 이루어져야 함(두개 포함하여)
  - 머신 word 크기 안이면 대부분의 플랫폼에서 atomic 연산을 통하여 변경가능

Implementing push() for a lock free queue

```
void push( T new_value )
{
    std::unique_ptr<T>    new_data( new T(new_value) );
    counted_node_ptr      new_next;

    new_next.ptr = new node;
    new_next.external_count = 1;
    counted_node_ptr old_tail = tail.load();

    for(;;)
    {
        increase_external_count( tail,old_tail );

        T*     old_data = nullptr;

        if( old_tail.ptr->data.compare_exchange_strong( old_data,
                                                        new_data.get() ) )
        {
            old_tail.ptr->next = new_next;
            old_tail=tail.exchange( new_next );
            free_external_counter( old_tail );
            new_data.release();
            break;
        }

        old_tail.ptr->release_ref();
    }
}
```

Poping a node from lock-free queue

```
std::unique_ptr<T> pop()
{
    counted_node_ptr old_head = head.load(std::memory_order_relaxed);

    for(;;)
    {
        increase_external_count( head,old_head );
        node* const ptr = old_head.ptr;

        if( ptr == tail.load().ptr )
        {
            ptr->release_ref();
            return std::unique_ptr<T>();
        }

        if( head.compare_exchange_strong( old_head,
                                          ptr->next ) )
        {
            T* const res = ptr->data.exchange( nullptr );
            free_external_counter( old_head );
            return std::unique_ptr<T>( res );
        }

        ptr->release_ref();
    }
}
```

Releasing a node reference

```
void release_ref()
{
    node_counter  old_counter = count.load(std::memory_order_relaxed);
    node_counter  new_counter;

    do
    {
        new_counter = old_counter;
        --new_counter.internal_count;
    }
    while( !count.compare_exchange_strong( old_counter,
                                           new_counter,
                                           std::memory_order_acquire,
                                           std::memory_order_relaxed ) );

    if( !new_counter.internal_count &&
        !new_counter.external_counters )
    {
        delete this;
    }
}
```

Obtaining a new reference to a node

```
static void increase_external_count( std::atomic<counted_node_ptr>&  counter,
                                     counted_node_ptr&               old_counter )
{
    counted_node_ptr new_counter;

    do
    {
        new_counter = old_counter;
        ++new_counter.external_count;
    } while( !counter.compare_exchange_strong( old_counter,
                                               new_counter,
                                               std::memory_order_acquire,
                                               std::memory_order_relaxed ) );

    old_counter.external_count = new_counter.external_count;
}
```

Freeing an external counter to a node

```
template<typename T>
class lock_free_queue
{
private:
    static void free_external_counter( counted_node_ptr &old_node_ptr )
    {
        node*          const ptr = old_node_ptr.ptr;
        int            const count_increase = old_node_ptr.external_count-2;
        node_counter   old_counter = ptr->count.load( std::memory_order_relaxed );
        node_counter   new_counter;

        do
        {
            new_counter = old_counter;
            --new_counter.external_counters;
            new_counter.internal_count += count_increase;
        } while( !ptr->count.compare_exchange_strong( old_counter,
                                                      new_counter,
                                                      std::memory_order_acquire,
                                                      std::memory_order_relaxed ) );

        if( !new_counter.internal_count &&
            !new_counter.external_counters )
        {
            delete ptr;
        }
    }
};
```

lock free ???

- race-free
- performance issue
- busy-wait 이 존재
  - cpu 자원 소모
  - 첫 쓰레드가 완료되기 전까지 다른 쓰레드는 작업 못함
- mutex 로 보호될 경우
  - cpu 가 다른 작업이 가능
- lock free 가 아니다???

```
void push( T new_value )
{
    std::unique_ptr<T>    new_data( new T(new_value) );
    counted_node_ptr      new_next;

    new_next.ptr = new node;
    new_next.external_count = 1;
    counted_node_ptr old_tail = tail.load();

    for(;;)
    {
        increase_external_count( tail,old_tail );

        T*     old_data = nullptr;

        if( old_tail.ptr->data.compare_exchange_strong( old_data,
                                                        new_data.get() ) )
        {
            old_tail.ptr->next = new_next;
            old_tail=tail.exchange( new_next );
            free_external_counter( old_tail );
            new_data.release();
            break;
        }

        old_tail.ptr->release_ref();
    }
}
```

Making The Queue Lock-free By Helping out Another Thread

- push() 할 쓰레드가 최대한 작업을 더 할수 있도록 한다.

- dummy 노드 를 만들어 tail 노드의 next 에 달아 놓고
- tail 포인터를 업데이트 한다.
- dummy 노드 를 생성한 쓰레드가 dummy 노드에 data 를 넣어도 되고
- 다른 쓰레드가 data 를 넣어도 된다.

pop() modified to allow helping on push() side

```
struct node
{
    std::atomic<T*>                 data;
    std::atomic<node_counter>       count;
    std::atomic<counted_node_ptr>   next;
};
```

pop() modified to allow helping on push() side

```
std::unique_ptr<T> pop()
{
    counted_node_ptr old_head = head.load( std::memory_order_relaxed );

    for(;;)
    {
        increase_external_count( head,old_head );

        node* const ptr = old_head.ptr;

        if( ptr == tail.load().ptr )
        {
            return std::unique_ptr<T>();
        }
        counted_node_ptr next = ptr->next.load();

        if( head.compare_exchange_strong( old_head,next ) )
        {
            T* const res = ptr->data.exchange( nullptr );
            free_external_counter( old_head );
            return std::unique_ptr<T>( res );
        }

        ptr->release_ref();
    }
}
```

- ptr->next 을 읽을시 묵시적으로 memory order 가 지정되었다.
- 그렇지만 memory_order_seq_cst_ordering 로 명시적으로 지정하는것이 도 좋다.

sample push() with helping for lock-free queue

```
void set_new_tail( counted_node_ptr        &old_tail,
                   counted_node_ptr const  &new_tail)
{
    node* const current_tail_ptr = old_tail.ptr;

    while( !tail.compare_exchange_weak( old_tail, new_tail) &&
           old_tail.ptr == current_tail_ptr );

    if( old_tail.ptr == current_tail_ptr )
        free_external_counter( old_tail );
    else
        current_tail_ptr->release_ref();
}
```

sample push() with helping for lock-free queue

```
void push( T new_value )
{
    std::unique_ptr<T>   new_data( new T( new_value ) );
    counted_node_ptr     new_next;

    new_next.ptr = new node;
    new_next.external_count = 1;

    counted_node_ptr old_tail = tail.load();

    for(;;)
    {
        increase_external_count( tail,old_tail );
        T*    old_data = nullptr;

        if( old_tail.ptr->data.compare_exchange_strong( old_data, new_data.get() ) )
        {
            counted_node_ptr     old_next={ 0 };

            if( !old_tail.ptr->next.compare_exchange_strong( old_next, new_next ) )
            {
                delete new_next.ptr;
                new_next = old_next;
            }

            set_new_tail( old_tail,
                          new_next );
            new_data.release();
            break;
        }
        else
        {
-  9 lines: counted_node_ptr      old_next = { 0 };----------------------------------
        }
    }
}
```

sample push() with helping for lock-free queue

```
void push( T new_value )
{
    std::unique_ptr<T>   new_data( new T( new_value ) );
    counted_node_ptr     new_next;

    new_next.ptr = new node;
    new_next.external_count = 1;

    counted_node_ptr old_tail = tail.load();

    for(;;)
    {
        increase_external_count( tail,old_tail );
        T*    old_data = nullptr;

        if( old_tail.ptr->data.compare_exchange_strong( old_data, new_data.get() ) )
        {
-- 12 lines: counted_node_ptr     old_next={ 0 };-------------------------------------
        }
        else
        {
            counted_node_ptr      old_next = { 0 };

            if( old_tail.ptr->next.compare_exchange_strong( old_next, new_next ) )
            {
                old_next = new_next;
                new_next.ptr = new node;
            }

            set_new_tail( old_tail, `old_next );
        }
    }
}
```

Handling multiple threads in push()

	Thread 1	Thread 2
	push(`A`)	push(`B`)

1	std::unique_ptr<T> new_data(new T(new_value));
	counted_node_ptr new_next;
	
	new_next.ptr = new node;
	new_next.external_count = 1;
	
	counted_node_ptr old_tail = tail.load();
	
	for(;;)
	{
	increase_external_count(tail,old_tail);
	T* old_data = nullptr;
	
	if(old_tail.ptr->data.compare_exchange_strong(
	old_data, new_data.get()))	

2		std::unique_ptr<T> new_data(new T(new_value));
		counted_node_ptr new_next;
		
		new_next.ptr = new node;
		new_next.external_count = 1;
		
		counted_node_ptr old_tail = tail.load();
		
		for(;;)
		{
		increase_external_count(tail,old_tail);
		T* old_data = nullptr;
		
		if(old_tail.ptr->data.compare_exchange_strong(
		old_data, new_data.get()))

queue

Node 1	<- HEAD <- TAIL

Node1	`A`, 0, nullptr
Node2	``, 0, nullptr

Thread 1	'A'
old_tail	TAIL
old_next	
new_next	1, Node2
Thread 2	'B'
old_tail	
old_next	
new_next	

Handling multiple threads in push()

	Thread 1	Thread 2
	push(`A`)	push(`B`)

1	`std::unique_ptr<T> new_data( new T( new_value ) );`
	`counted_node_ptr new_next;`
	
	`new_next.ptr = new node;`
	`new_next.external_count = 1;`
	
	`counted_node_ptr old_tail = tail.load();`
	
	`for(;;)`
	`{`
	`  increase_external_count( tail,old_tail );`
	`  T* old_data = nullptr;`
	
	`  if( old_tail.ptr->data.compare_exchange_strong(`
	`  old_data, new_data.get() ) )`	

2		`std::unique_ptr<T> new_data( new T( new_value ) );`
		`counted_node_ptr new_next;`
		
		`new_next.ptr = new node;`
		`new_next.external_count = 1;`
		
		`counted_node_ptr old_tail = tail.load();`
		
		`for(;;)`
		`{`
		`  increase_external_count( tail,old_tail );`
		`  T* old_data = nullptr;`
		
		`  if( old_tail.ptr->data.compare_exchange_strong(`
		`  old_data, new_data.get() ) )`

queue

Node 1	<- HEAD <- TAIL

Node1	`A`, 0, nullptr
Node2	``, 0. nullptr
Node3	``, 0, nullptr

Thread 1	'A'
old_tail	TAIL
old_next	
new_next	1, Node2
Thread 2	'B'
old_tail	TAIL
old_next	
new_next	1, Node3

Handling multiple threads in push()

	Thread 1	Thread 2
	`push( `A` )`	`push( `B` )`
1	`if( old_tail.ptr->data.compare_exchange_strong( old_data, new_data.get() ) )`	
2		`if( old_tail.ptr->data.compare_exchange_strong( old_data, new_data.get() ) )`
3		`else { counted_node_ptr old_next = { 0 }; if( old_tail.ptr->next.compare_exchange_strong( old_next, new_next ) )`
4		`{ old_next = new_next; new_next.ptr = new node; } set_new_tail( old_tail, old_next ); }`
5	`counted_node_ptr old_next = { 0 }; if( !old_tail.ptr->next.compare_exchange_strong( old_next, new_next ) ) { delete new_next.ptr; new_next = old_next; } set_new_tail( old_tail, new_next ); new_data.release(); } break;`	

queue

Node 1	<- HEAD <- TAIL

Node1	`A`, **1, Node3**
Node2	``, 0, nullptr
Node3	``, 0, nullptr

Thread 1	'A'
old_tail	TAIL
old_next	
new_next	1, Node2
Thread 2	'B'
old_tail	TAIL
old_next	0, nullptr
new_next	1, Node3

Handling multiple threads in push()

	Thread 1	Thread 2
	push(`A`)	push(`B`)
1	`if( old_tail.ptr->data.compare_exchange_strong( old_data, new_data.get() ) )`	
2		`if( old_tail.ptr->data.compare_exchange_strong( old_data, new_data.get() ) )`
3		`else { counted_node_ptr old_next = { 0 }; if( old_tail.ptr->next.compare_exchange_strong( old_next, new_next ) )`
4		`{ old_next = new_next; new_next.ptr = new node; } set_new_tail( old_tail, old_next ); }`
5	`counted_node_ptr old_next = { 0 }; if( !old_tail.ptr->next.compare_exchange_strong( old_next, new_next ) ) { delete new_next.ptr; new_next = old_next; } set_new_tail( old_tail, new_next ); new_data.release(); } break;`	

queue

Node 1	<- HEAD
Node3	<- TAIL

Node1	`A`, 1, Node3
Node2	``, 0, nullptr
Node3	``, 0, nullptr
Node4	``, 0, nullptr

Thread 1	'A'
old_tail	TAIL
old_next	
new_next	1, Node2
Thread 2	'B'
old_tail	TAIL
old_next	1, Node3
new_next	1, Node4

Handling multiple threads in push()

	Thread 1	Thread 2
	push(`A`)	push(`B`)
1	`if( old_tail.ptr->data.compare_exchange_strong( old_data, new_data.get() ) )`	
2		`if( old_tail.ptr->data.compare_exchange_strong( old_data, new_data.get() ) )`
3		`else { counted_node_ptr old_next = { 0 }; if( old_tail.ptr->next.compare_exchange_strong( old_next, new_next ) )`
4		`{ old_next = new_next; new_next.ptr = new node; } set_new_tail( old_tail, old_next ); }`
5	`counted_node_ptr old_next = { 0 }; if( !old_tail.ptr->next.compare_exchange_strong( old_next, new_next ) ) { delete new_next.ptr; new_next = old_next; } set_new_tail( old_tail, new_next ); new_data.release(); } break;`	

queue

Node 1	<- HEAD
Node3	<- TAIL

Node1	`A`, 1, Node3
Node2	``, 0, nullptr
Node3	``, 0, nullptr
Node4	``, 0, nullptr

Thread 1	'A'
old_tail	TAIL
old_next	0, nullptr
new_next	0, nullptr
Thread 2	'B'
old_tail	TAIL
old_next	
new_next	1, Node4

Handling multiple threads in push()

	Thread 1	Thread 2
	`push( `A` )`	`push( `B` )`
4		`set_new_tail( old_tail, old_next );` `}`

5	`counted_node_ptr old_next = { 0 };`
	`if( !old_tail.ptr->next.compare_exchange_strong(`
	`old_next, new_next ) )`
	`{`
	`delete new_next.ptr;`
	`new_next = old_next;`
	`}`
	
	`set_new_tail( old_tail, new_next );`
	`new_data.release();`
	`}`
	`break;`	

6		`increase_external_count( tail, old_tail );`
		`T* old_data = nullptr;`
		
		`if( old_tail.ptr->data.compare_exchange_strong(`
		`old_data, new_data.get() ) )`
		`{`

7		`counted_node_ptr old_next = { 0 };`
		`if( !old_tail.ptr->next.compare_exchange_strong(`
		`old_next, new_next ) )`
		`{`
		`delete new_next.ptr;`
		`new_next = old_next;`
		`}`
		
		`set_new_tail( old_tail, new_next );`
		`new_data.release();`
		`}`
		`break;`

queue

Node 1 <- HEAD
Node3 <- TAIL

Node1	`A`, 1, Node3
Node2	``, 0, nullptr
Node3	`B`, 0, nullptr
Node4	``, 0, nullptr

Thread 1	'A'
old_tail	TAIL
old_next	0, nullptr
new_next	0, nullptr
Thread 2	'B'
old_tail	TAIL
old_next	
new_next	1, Node4

Handling multiple threads in push()

	Thread 1	Thread 2
	push(`A`)	push(`B`)
4		set_new_tail(old_tail, old_next); }

5	counted_node_ptr old_next = { 0 };
	if(!old_tail.ptr->next.compare_exchange_strong(
	old_next, new_next))
	{
	delete new_next.ptr;
	new_next = old_next;
	}
	
	set_new_tail(old_tail, new_next);
	new_data.release();
	}
	break;	

6		increase_external_count(tail, old_tail);
		T* old_data = nullptr;
		
		if(old_tail.ptr->data.compare_exchange_strong(
		old_data, new_data.get()))
		{

7		counted_node_ptr old_next = { 0 };
		if(!old_tail.ptr->next.compare_exchange_strong(
		old_next, new_next))
		{
		delete new_next.ptr;
		new_next = old_next;
		}
		
		set_new_tail(old_tail, new_next);
		new_data.release();
		}
		break;

queue

Node 1	<- HEAD
Node3	
Node4	<- TAIL

Node1	`A`, 1, Node3
Node2	``, 0, nullptr
Node3	`B`, 1, Node4
Node4	``, 0, nullptr

Thread 1	'A'
old_tail	TAIL
old_next	0, nullptr
new_next	0, nullptr
Thread 2	'B'
old_tail	TAIL
old_next	0, nullptr
new_next	1, Node4

Performance Test (시나리오)

- Thread 별 1,000,000번 push 완료후 1,000,000 번 pop 수행

```
void test( int                                aThreadID,
           lock_free_queue<std::string> * aLockFreeQueue,
           int                                aCount )
{

    for ( int i = 0; i < aCount; i++ )
    {
        aLockFreeQueue->push( "TEST1234567890" );
    }

    for ( int i = 0; i < aCount; i++ )
    {
        aLockFreeQueue->pop();
    }
}
```

Performance Test (시나리오)

- Thread 수 늘려가며 수행 시간 및 cpu 사용량 측정
  - time 명령어 사용

```
TIME(1)                    BSD General Commands Manual                    TIME(1)

NAME
     time -- time command execution

SYNOPSIS
     time [-lp] utility

DESCRIPTION
     The time utility executes and times utility.  After the utility finishes, time writes the
     total time elapsed, the time consumed by system overhead, and the time used to execute
     utility to the standard error stream.  Times are reported in seconds.
```

```
 DoO@DooSeonui-MacBook-Air ~/Work/Study/Whatever/LockFreeQueue/build  master ●  time
shell  0.09s user 0.09s system 5% cpu 3.440 total
children  0.11s user 0.20s system 8% cpu 3.440 total
```

User	User Mode 에서 CPU 를 사용한 시간
sysem	Kernel Mode 에서 CPU 를 사용한 시간
total	총 CPU 를 사용한 시간

Performance Test (Test 환경)

- Test 장비

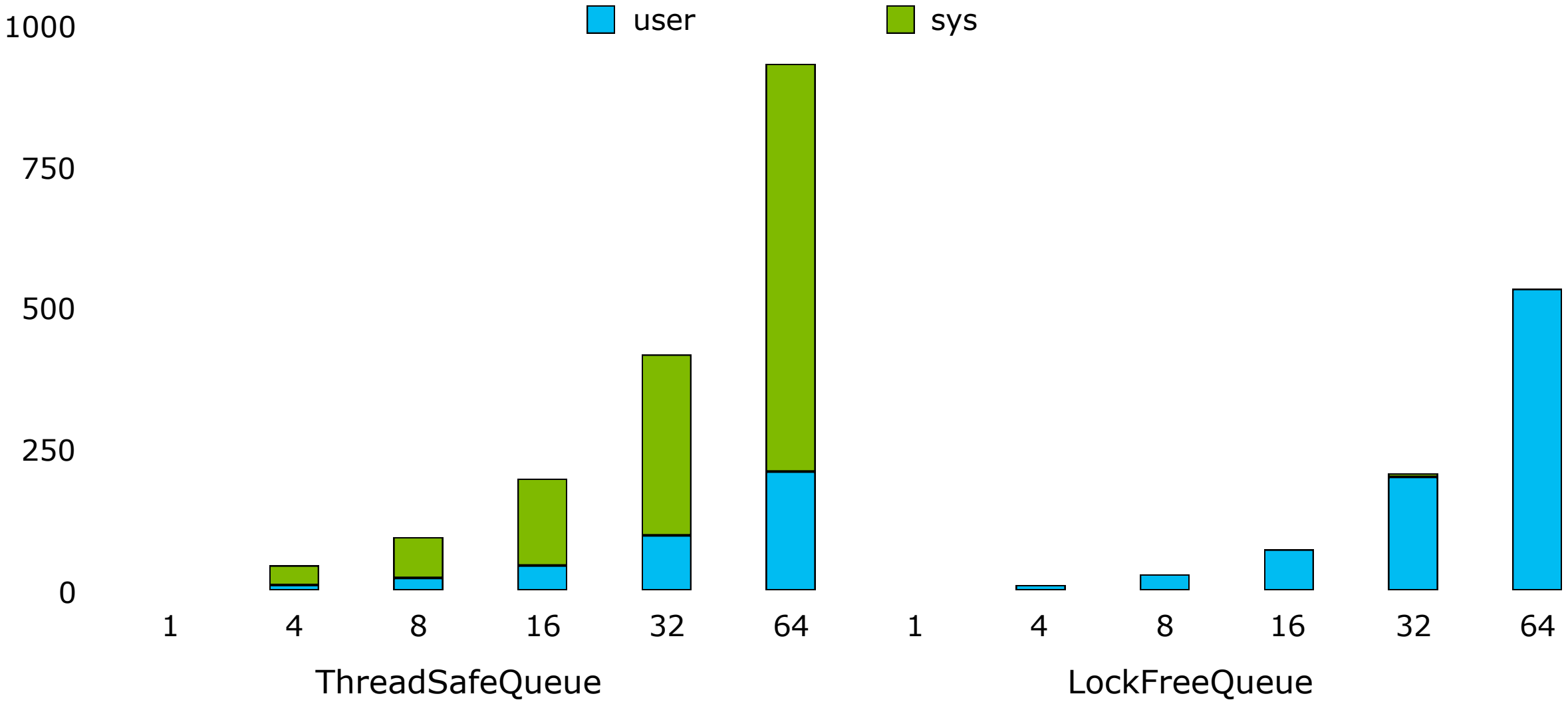
- 컴파일러

```
$ g++ -v
Configured with: --prefix=/Applications/Xcode.app/Contents/Developer/usr --with-gxx-include-dir=/usr/
include/c++/4.2.1
Apple LLVM version 7.0.2 (clang-700.1.81)
Target: x86_64-apple-darwin15.3.0
Thread model: posix
```

Performance Test (수행시간)

		1	4	8	16	32	64
ThreadSafeQueue	user	0.72	11.88	23.25	47.08	97.53	211.40
	sys	0.04	34.70	72.09	150.95	322.91	723.82
	real	0.771	41.083	84.63	174.07	361.09	780
LockFreeQueue	user	0.82	11.91	29.05	74.84	201.91	537.76
	sys	0.04	0.2	0.4	0.85	5.69	32.26
	real	0.86	3.23	7.43	19.249	53.122	155.63

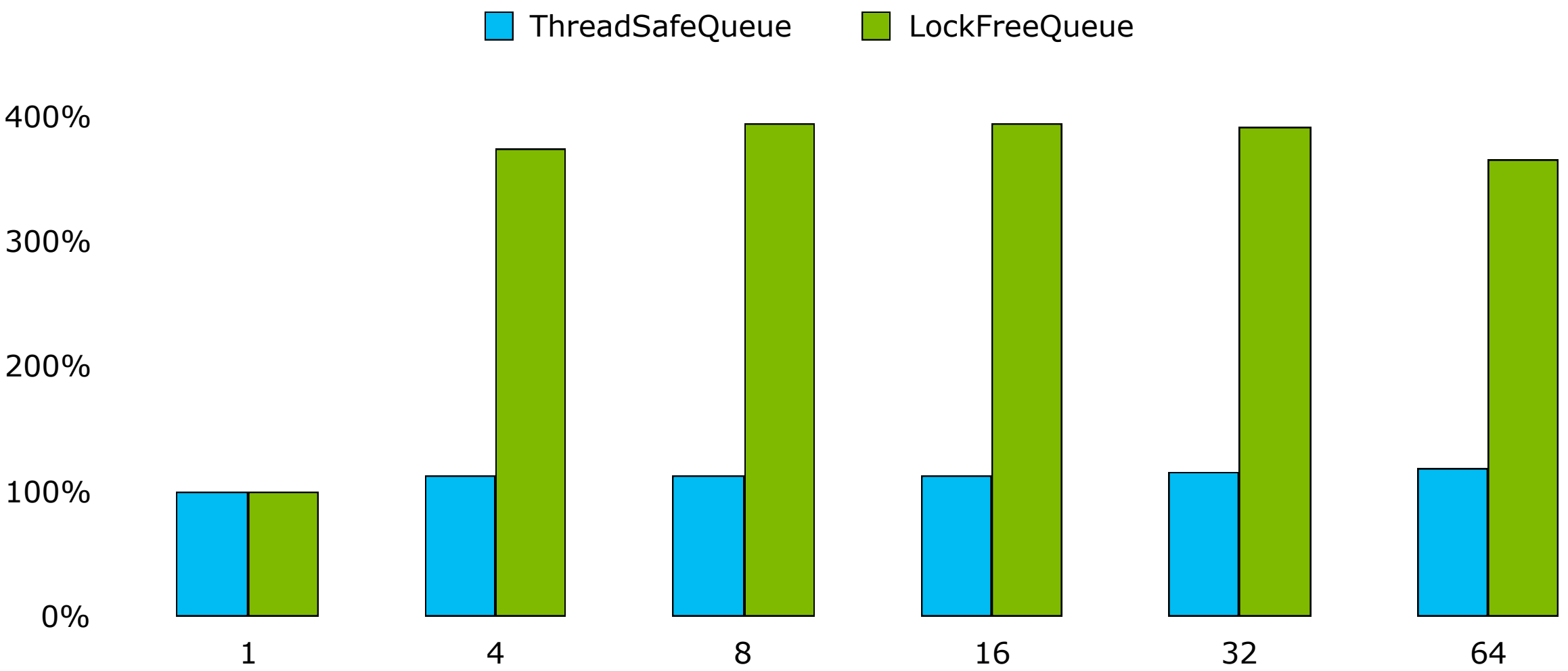
Performance Test (수행시간)



Performance Test (CPU 사용량)

	1	4	8	16	32	64
ThreadSafeQueue	99%	113%	112%	113%	116%	119%
LockFreeQueue	99%	374%	394%	393%	390%	365%

Performance Test (CPU 사용량)



Performance Test (Test 환경)

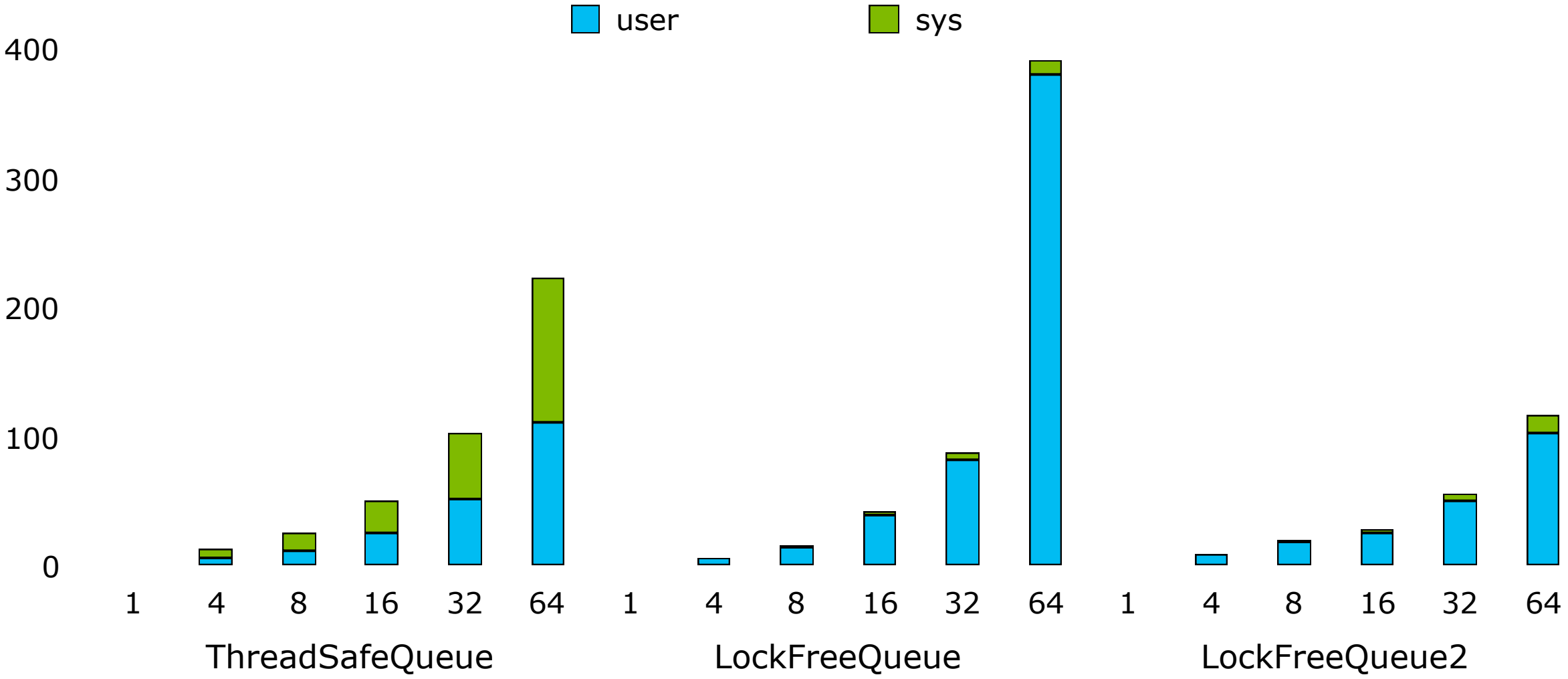
- Test 장비
  - CPU
    - Intel(R) Core(TM) i5-3470 CPU @ 3.20GHz
    - 4 Core
  - Memroy
    - 32 GB
- 컴파일러

```
$ g++ -v
Using built-in specs.
COLLECT_GCC=g++
COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-linux-gnu/4.8/lto-wrapper
Target: x86_64-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu 4.8.4-2ubuntu1~14.04' --with-bugurl=file:///usr/share/doc/gcc-4.8/README.Bugs
--enable-languages=c,c++,java,go,d,fortran,objc,obj-c++ --prefix=/usr --program-suffix=-4.8 --enable-shared --enable-linker-build-id --
libexecdir=/usr/lib --without-included-gettext --enable-threads=posix --with-gxx-include-dir=/usr/include/c++/4.8 --libdir=/usr/lib --enable-
nls --with-sysroot=/ --enable-clocale=gnu --enable-libstdcxx-debug --enable-libstdcxx-time=yes --enable-gnu-unique-object --disable-
libmudflap --enable-plugin --with-system-zlib --disable-browser-plugin --enable-java-awt=gtk --enable-gtk-cairo --with-java-home=/usr/lib/
jvm/java-1.5.0-gcj-4.8-amd64/jre --enable-java-home --with-jvm-root-dir=/usr/lib/jvm/java-1.5.0-gcj-4.8-amd64 --with-jvm-jar-dir=/usr/lib/
jvm-exports/java-1.5.0-gcj-4.8-amd64 --with-arch-directory=amd64 --with-ecj-jar=/usr/share/java/eclipse-ecj.jar --enable-objc-gc --enable-
multiarch --disable-werror --with-arch-32=i686 --with-abi=m64 --with-multilib-list=m32,m64,mx32 --with-tune=generic --enable-checking=release
--build=x86_64-linux-gnu --host=x86_64-linux-gnu --target=x86_64-linux-gnu
Thread model: posix
gcc version 4.8.4 (Ubuntu 4.8.4-2ubuntu1~14.04)
```

Performance Test (수행시간)

		1	4	8	16	32	64
ThreadSafeQueue	user	0.98	7.53	12.85	26.29	52.47	112.23
	sys	0.03	6.52	14.01	24.53	50.38	111.34
	real	1.013	4.727	8.99	18.80	39.62	77.57
LockFreeQueue	user	0.52	7.08	15.52	39.81	82.67	380.45
	sys	0.06	0.27	1.17	2.6	5.32	11.61
	real	0.538	2.291	5.87	16.07	35.83	129.21
LockFreeQueue2	user	0.58	9.73	18.97	26.69	50.52	103.25
	sys	0.06	0.34	1.51	2.90	5.88	13.70
	real	0.644	3.031	6.681	12.813	26.40	44.34

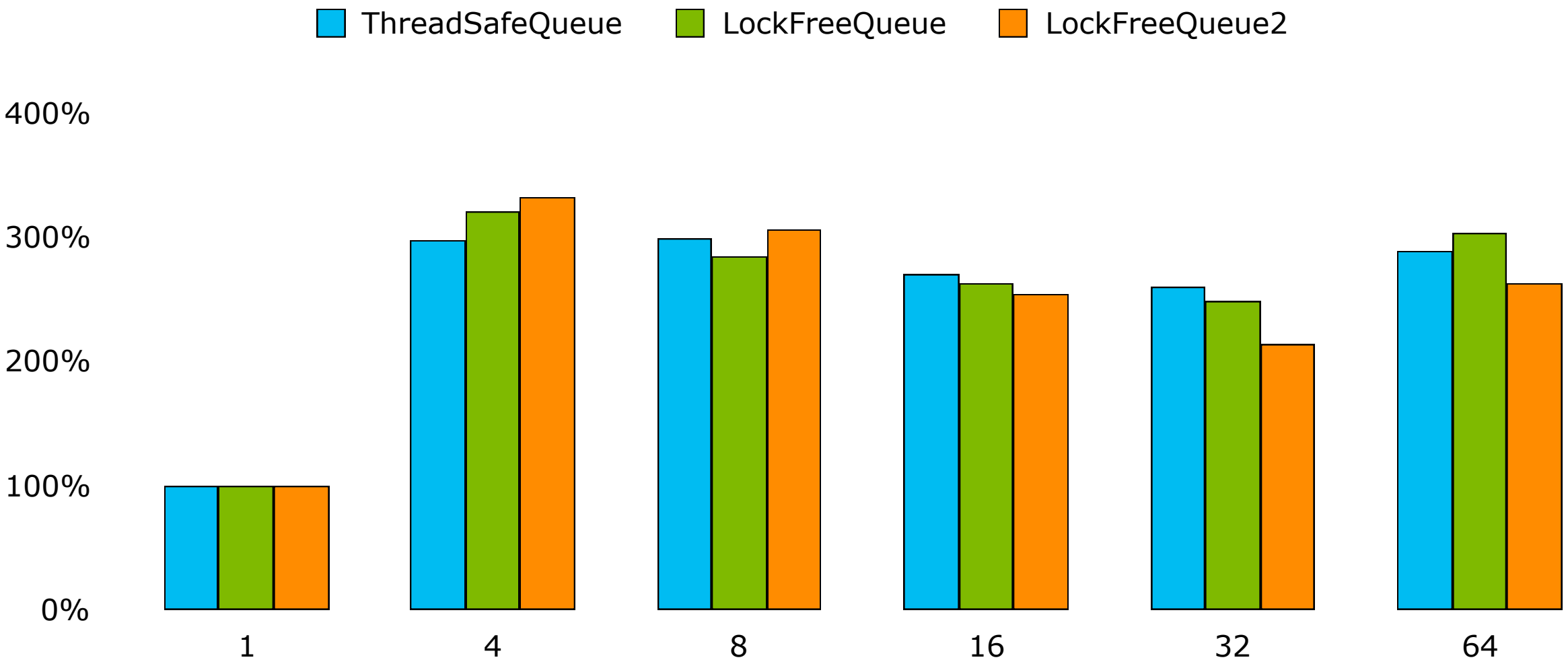
Performance Test (수행시간)



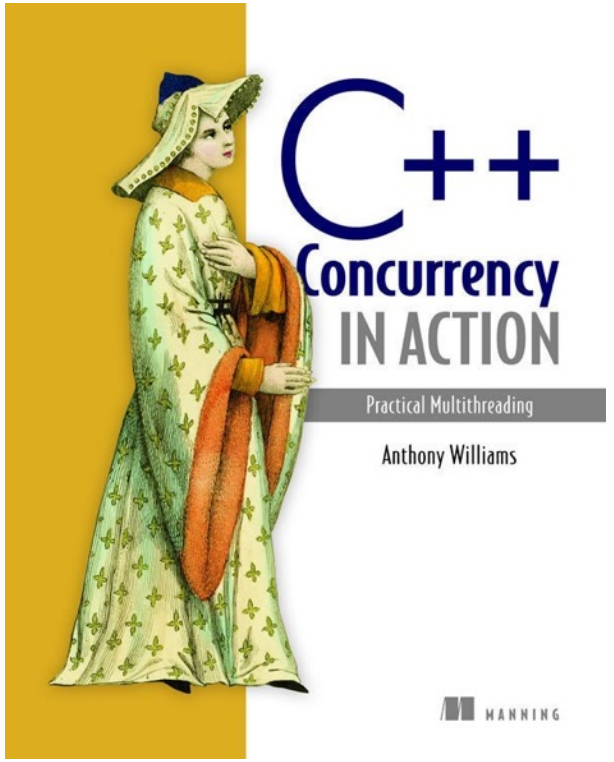
Performance Test (CPU 사용량)

	1	4	8	16	32	64
ThreadSafeQueue	99%	297%	298%	270%	259%	288%
LockFreeQueue	99%	320%	284%	263%	248%	303%
LockFreeQueue2	99%	332%	306%	254%	213%	263%

Performance Test (CPU 사용량)



Guidelines for writing lock free data structure



use std::memory_order_seq_cst for prototyping

- std::memory_order_seq_cst
  - 모든 연산의 순서를 보장

- 일반적으로 자료구조의 근본이 되는 연산을 수행하는 모든 코드를 확인할수 있을 때 제한을 완화시킬수 있을지 결정할수 있다.
- Attempting to do otherwise just makes your life harder

- 코드가 당장 동작하더도 정상동작인지는 보장할수 없다.
- 모든 상황을 확인할수 있는 검사툴이 없는 이상 지정된 ordering 으로 동작여부를 보장할수 없다.

use a lock-free memory reclamation scheme

- lock free 코드에서 메모리 관리는 가장 어려운 이슈중에 하나
  - 다른 쓰레드가 참조 하고 있는 객체를 삭제 하지 않는것이 필수
  - 그러나 과도한 메모리 사용을 자제하기 위해서 가능한 빨리 객체를 삭제

- 이 챕터에서는 메모리를 안전하게 사용을 위해서 아래 3가지 기술들에 대해서 보았다.
  - 자료구조에 접근하는 쓰레드가 없을때까지 대기하고 대기하고 있는 모든 객체를 삭제
  - 쓰레드가 특정한 객체에 접근하는 것을 식별하기 위하여 hazard 포인터를 사용
  - 객체가 참조되고 있을때 삭제가 되지않도록 객체의 reference counting 을 사용

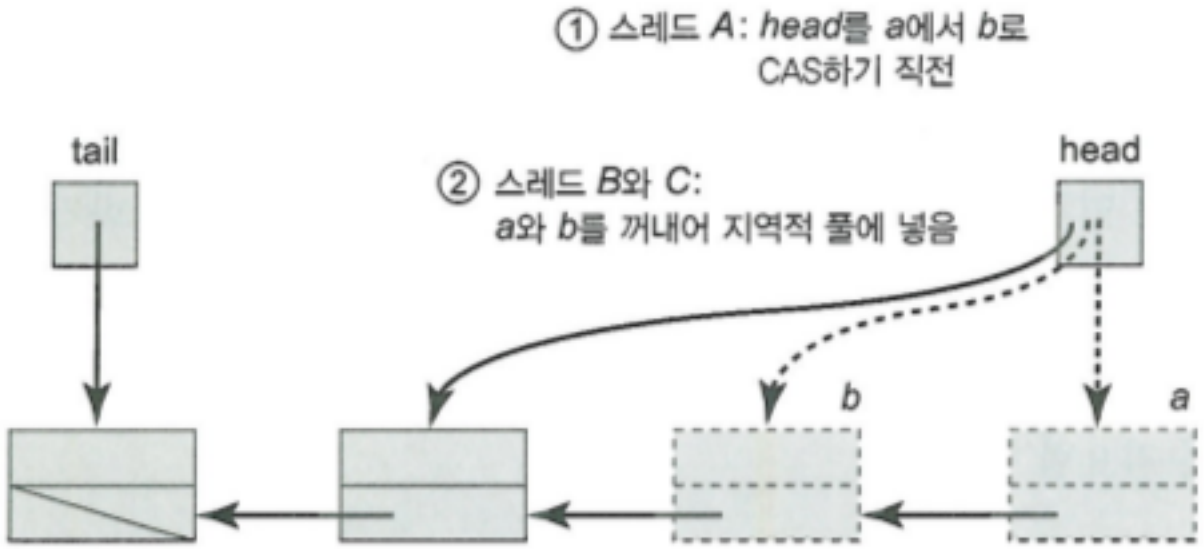
- 이상적인 시나리오는 garbage collector 를 사용
  - garbage collector가 더이상 사용되지 않는 노드를 자동으로 해제
  - 그렇기 때문에 lock-free 알고리즘을 작성하기 쉽다.

- node 를 재활용하고 자료구조를 없앨때 node 을 삭제
  - 메모리를 재 사용하기 때문에 유효하지 않은 메모리를 사용하지 않음
  - ABA Problem 을 야기

watch out for ABA Problem

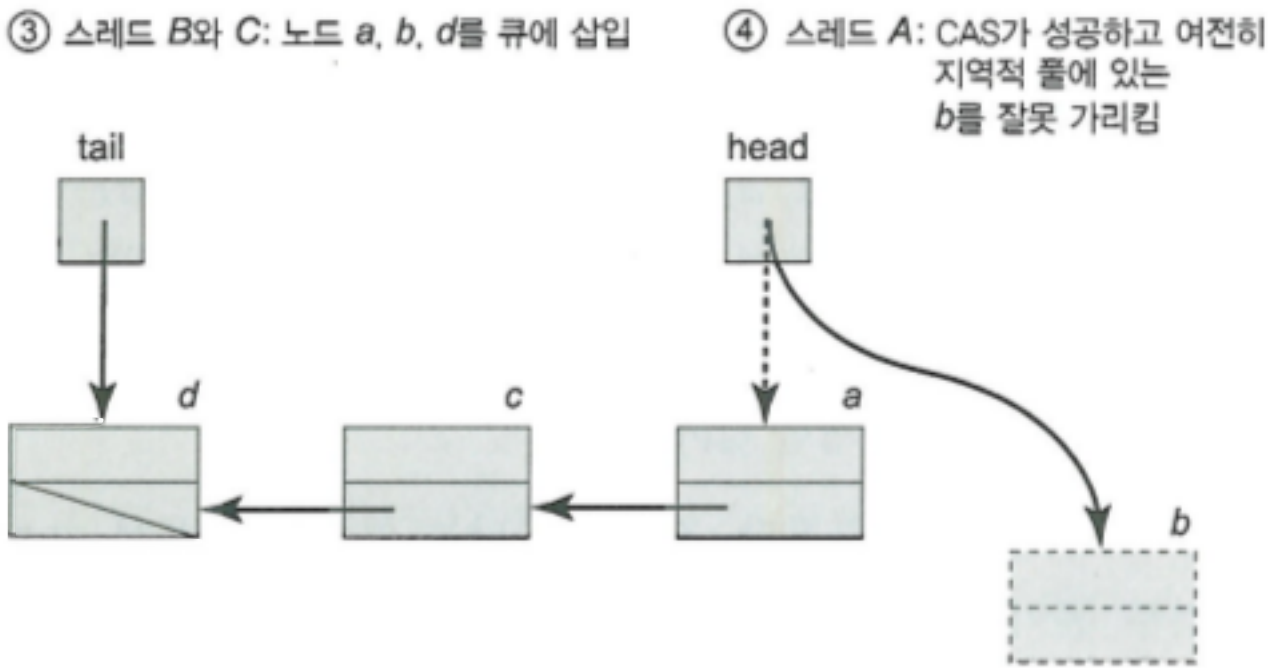
- 1. T1은 atomic 으로 변수 x 를 읽고 x 에는 A 값이 저장되어 있음
- 2. T1은 A값으로 연산을 수행
- 3. T1은 OS에 의하여 중지, T2를 수행
- 4. T2는 변수 x을 B로 값을 변경
- 5. T1에서의 A값은 더이상 유효하지 않음
- 6. T3는 새로운 데이터를 가지고 변수 x 를 다시 A로 값을 변경
- 7. T1이 다시 수행 x 에 compare/exchange로 값 A를 비교

watch out for ABA Problem



From The Art of Multiprocessor Programming

watch out for ABA Problem



From The Art of Multiprocessor Programming

watch out for ABA Problem

	Thread1	Thread2	Thread3	Thread4
1	POP()			
2		POP()		
3				PUSH()
4		PUSH()		
5			POP()	
7	head.CAS(A, B)			

Queue	
A	Head
B	Tail
Free Memory List	
C	
D	
E	

watch out for ABA Problem

	Thread1	Thread2	Thread3	Thread4
1	POP()			
2		POP() -> free A		
3				PUSH()
4		PUSH()		
5			POP()	
7	head.CAS(A, B)			

Queue	
B	Head Tail
Free Memory List	
A	
C	
D	
E	

watch out for ABA Problem

	Thread1	Thread2	Thread3	Thread4
1	POP()			
2		POP() -> free A		
3				PUSH() -> new A
4		PUSH()		
5			POP()	
7	head.CAS(A, B)			

Queue	
B	Head
A	Tail
Free Memory List	
C	
D	
E	

watch out for ABA Problem

	Thread1	Thread2	Thread3	Thread4
1	POP()			
2		POP() -> free A		
3				PUSH() -> new A
4		PUSH() -> new C		
5			POP()	
7	head.CAS(A, B)			

Queue	
B	Head
A	
C	Tail
Free Memory List	
D	
E	

watch out for ABA Problem

	Thread1	Thread2	Thread3	Thread4
1	POP()			
2		POP() -> free A		
3				PUSH() -> new A
4		PUSH() -> new C		
5			POP() -> free B	
7	head.CAS(A, B)			

Queue	
A	
C	Tail
Free Memory List	
B	Head
D	
E	

watch out for ABA Problem

- 해결방법
  - 변수와 ABA counter 를 같이 사용
  - compare/exchange 연산을 ABA counter 와 변수를 합쳐서 atomic 하게 수행
  - 변수의 값이 변경되지 않았어도 ABACounter 값이 변경되었기 때문에 ABA Problem 을 회피

- 메모리 할당자에서 free list 를 사용하거나 노드를 재활용하는 알고리즘에서 많이 발생

identify busy-wait loops and help the other thread-safe

- busy wait loop
  - 마지막 queue 의 예제에서 push() 의 연산은 다른 쓰레드의 push() 연산의 완료를 대기
  - 대기 하는 쓰레드는 진행을 못하고 CPU 자원을 사용하여 push() 연산의 완료를 확인/대기

- busy-wait loop 이 있으면 mutex 와 lock 같은 blocking 연산이 더 효율적
  - busy-wait loop 은 cpu 자원 소모
  - mutex 는 적어도 cpu 자원을 소모 안함

- busy-wait loop 제거
  - 데이터 멤버를 non-atomic 변수에서 atomic 변수로 변경
  - 데이터 멤버의 값을 변경을 위해서 compare/exchage 연산이 필요
  - 더 복잡한 자료구조 와 더 큰 구조적 변경이 필요

mutex vs spin lock

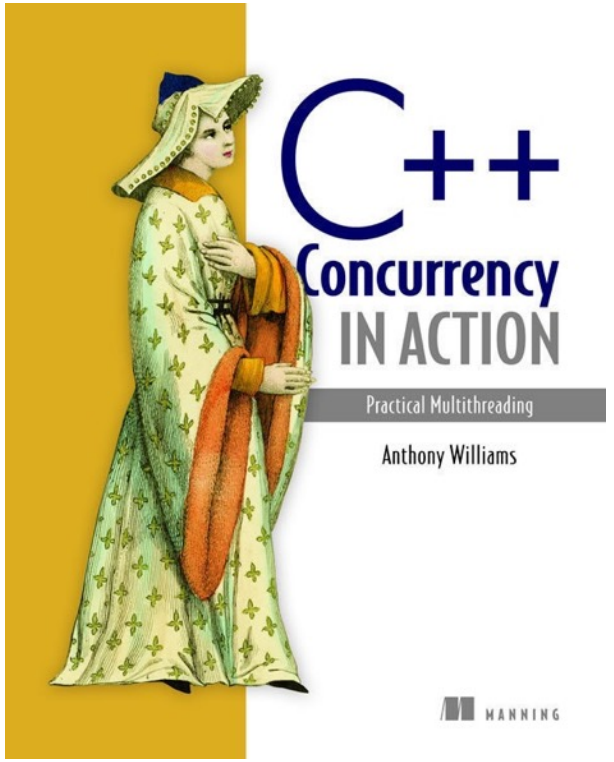
- mutex
  - 대기 하는 동안 cpu 자원 소모하지 않음
  - system call
  - 운영체제가 스케줄링 가능
  - 검증 되어 있음(운영체제 수준에서)
  - 경합이 많이 발생하는 자료구조에 유리

- spin lock (busy-wait loop)
  - 대기 하는 동안 cpu 자원 소모
  - system call 이 아님
  - 운영체제가 스케줄링 불 가능
  - 검증이 어려움
  - 경합이 많이 발생하지 않는 자료구조에 유리

system call

- 강제 cache 무효화
- user mode 와 kernel mode 전환 (context switching)
- 위와 같은 이유로 성능 하락

SUMMARY


C++
Concurrency
IN ACTION
Practical Multithreading
Anthony Williams
MANNING

Summary

- 확인한 lock-free 자료구조
  - lock-free stack
  - lock-free queue

- atomic 연산을 통하여 memory ordering 에 주의
  - data race 가 존재하지 않아야함
  - 각 쓰레드가 자료구조의 일관된 view를 봐야함

- lock-free 자료구조의 설계는 어렵고 실수하기가 쉽다.

- 쓰레드 사이에 공유되 데이터가 존재
  - 쓰레드 사이에서 어떻게 데이터를 동기화 하는 방법
  - 사용해야할 자료구조

- concurrency 자료구조를 추상화 하여 남은 코드는 데이터 동기화 보다는 데이터에 집중

앞으로는...?

- Transaction Memory
  - DB 의 Transaction 을 Memory 수준에서 지원
  - Hardware Transaction Memory
    - Intel - TSX (Transaction Synchronization Extensions)
    - AMD - ASF (Advanced Synchronized Facility)
  - Software Transaction Memory
  - 보다 쉽게 lock free 알고리즘 구현 가능

- Hardware Lock Elision
  - Compile 시간에 mutex 를 HLE 로 대체
  - 재코딩 없이 컴파일러만 변경시 적용

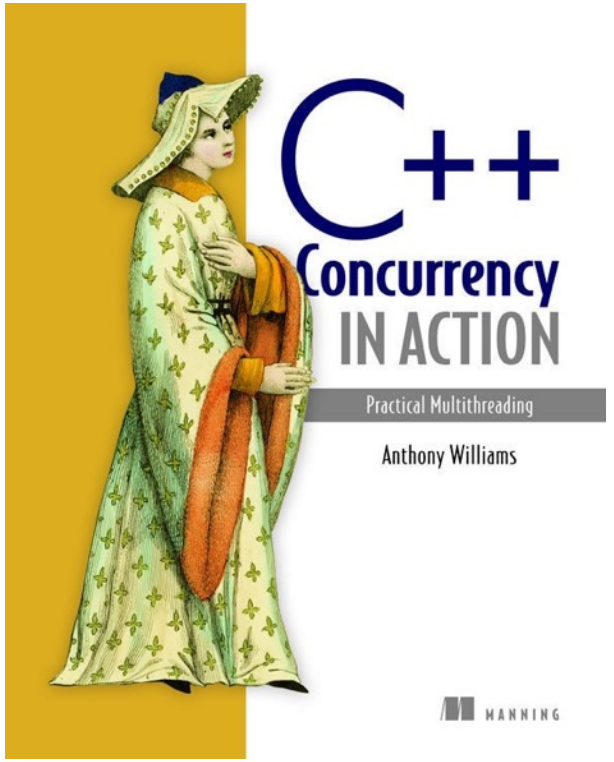
Conclusion

- lock-free 는 어렵다….
  - 디자인 하기 어렵다..
  - 구현 하기 어렵다..
  - 테스트 코드 작성이 어렵다.
  - 검증 하기 어렵다.
- 쓰지 말자…?
- 만약 써야 한다면..
  - 철저히 리뷰를 하자.
  - 모든 동시성 문제를 생각해보자.
  - 테스트 코드를 반드시 작성하자. 그래도 모든것이 테스트 했다고 생각하지 말자.
  - 논문등을 통하여 검증된 알고리즘을 구현하자. (??? tlsf 메모리 관리자 등등)

Conclusion

- 꼭 구현해야 되나??
  - 이미 구현되어 있는 좋은 Container 들이 많다.
    - Intel TBB Container
    - Boost
    - libcds

Q & A


C++
Concurrency
IN ACTION
Practical Multithreading
Anthony Williams
MANNING

reference

- Concurrency In Action
- The Art of Multiprocessor Programming
- https://forums.manning.com/
- http://en.cppreference.com/
- intel.com
- http://libcds.sourceforge.net
- https://www.threadingbuildingblocks.org
- https://www.threadingbuildingblocks.org/intel-tbb-tutorial
- http://www.boost.org/doc/libs/1_60_0/doc/html/lockfree.html